Visualization of Dispatching Strategies

Semester Thesis

Michael Grossniklaus
<michael@vis.ethz.ch>

Prof. Dr. Moira C. Norrie
Supervisor: Christian Bach

Global Information Systems Group
Institute for Information Systems
Department of Computer Science

July 2000

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

globis

Preface

This document will help you to get an insight of what VISUAL DISPATCHER does, how it works and how you can extend it to do what you want it to do.
The document is partitioned into four major sections. Each section can be read independently from the others. First you will find a short introduction to the topic of my semester work and a reminiscence about the development of VISUAL DISPATCHER.
The second chapter provides a quick user manual explaining what can be done with VISUAL DISPATCHER and how it is done. If you just want to experiment with the dispatching strategies of JAVA and LOOTION, then chapter two is what you should read.
In chapter three, there will be a discussion of the software architecture of VISUAL DISPATCHER. One major goal of this project was to keep the architecture of the software open, so that later on other dispatching strategies of different programming languages could be integrated. In this third section one finds a documentation of the software and hints, how to extend VISUAL DISPATCHER.
Finally there is a forth short section containing some possible future works, that can be done to improve or extend VISUAL DISPATCHER. Those ideas are mostly things that came to my mind during the development of the application and given more time I would have integrated them myself.

Michael Grossniklaus, July 2000

Contents

1 Introduction

The aim of this semester work was to develop a visualisation tool to explore dispatching strategies of different object-oriented programming languages. As the process of method resolution (the dispatching of messages) is a central distinguishing property of object-oriented programming languages the tool should allow to explore and understand a language's dispatching strategy in an intuitive way.
As a part of this project the tool had to be able to simulate the behaviour of JAVA [2] and LOOTION [6], an object-oriented programming language developed at the Institute of Global Information Systems at ETH Zurich. Furthermore extensibility was an important aspect of the project. The tool should be designed to allow the integration of other language's dispatching strategies in the future.

1.1 Research and Concept

Never really having given a thought about how object-oriented programming languages do method resolution, one isn't aware of the possibility, that there could be more than one way to do this. So when starting this project it was necessary to determine how vast the field of possibilities is, to be able to develop a software that could deal with future dispatching strategies completely unknown today. To this end information about JAVA [2, 3, 4] and LOOTION [6] had to be gathered. In doing so one can observe, that all object-oriented programming languages are built out of the same components: classes as prototypes of objects and inheritance constraints specifying subclass relationships. What was missing at that point was an idea, how to visualize this.
In his paper about CECIL [1], an object-oriented programming language capable of multiple inheritance and multi-methods, Chambers is facing the same challange of visualizing his explanations of CECIL's dispatching strategy with illustrations, that empower the user to replay and understand the steps of the algorithm. Using his method seemed to be a sound idea for this project too.
When talking about dispatching, there are two important aspects, that have to be included in

a graphical figure of a given situation. On the one side there is the inheritance relationship in which the classes stand to each other, defining which methods are accessible to a given class in the graph. On the other side one has to have a way of displaying the methods of the classes.

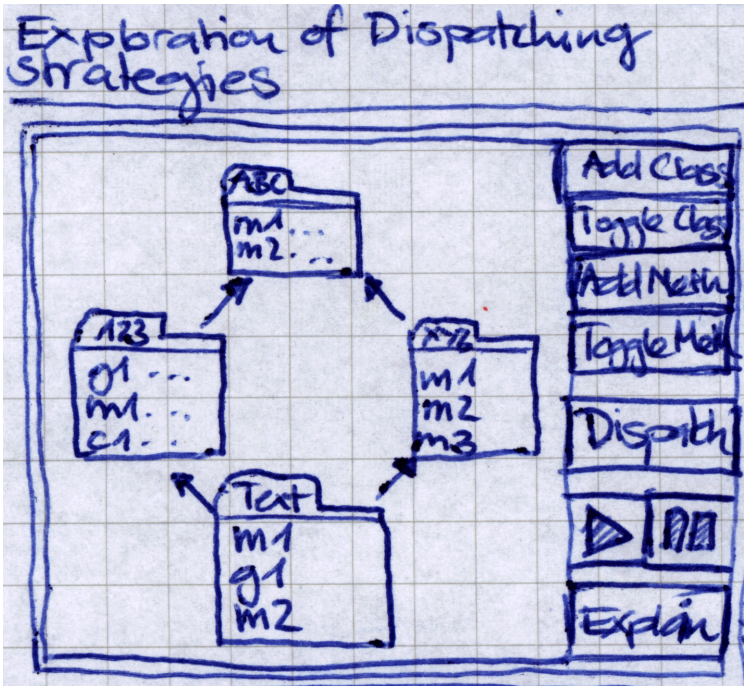

Figure 1.1: Early Concept of the Main Window

As a consequence it is not possible to primarily display the inheritance structure and having the user keep the record, which method was declared in what class. But on the other hand it is absolutely not necessary to display the entire class bodies in an enormous inheritance graph. It was probably at this time when a first sketch (Figure 1.1) of how the application could look like came up.

1.2 Graphical User Interface

Having found a concept for the graphical user interface, experiments with the SWING toolkit started. When one has no experience working with such a toolkit, it is usually best to try different approaches to implement the desired object. This technique soon led to a working component based on `JDesktop` that could visualize classes, but it was not possible to incorporate the arrows for subclass relationships. Obviously using `JDesktop` had been a mistake since it didn't work well and provided functionality that would never be used. Probably the best solution was to develop own extensions of `JComponent` and thereby building custom SWING objects to suit the purposes of this application.
With absolutely no idea how SWING is built and working internally, one has to do some reading before engaging in such an endeavor. "Graphic Java - Volume II" by David M. Geary [5] is a very useful book for this matter, providing everything from programming tips to theoretical background to reference. Having studied the first few chapters on SWING

architecture, it became feasible to implement the envisioned components with everything wished for!

1.3 Java Dispatching

At this time the front-end worked satisfactorily and it could be regarded as a good sign, that it was developed with no particular programming language in mind. To test the theory, that it would work to visualize any given object-oriented language, first steps towards implementing a dispatching engine were taken. Since there was no information about LOOTION's dispatching strategy available at that time, focus was given to JAVA first.
First an adequate data structure to hold all information the user entered via the graphical interface had to be developed. From the viewer structure it was clear that there had to be two main components in this data structure. But although there were plans to use the same viewers for all programming languages, it was not possible to use this approach for the underlying data structure as far as classes are concerned. Therefore an abstract class with a predefined interface was implemented on which the viewer classes would operate on. How to use this class to model different languages is shown in a later chapter.
As a next thing a set of dialogue classes was built empowering the user to enter all information of his design in a graphical way. Since such classes are not as easy to extend as the classes of the data structure, a lot of thought was given to self-adapting dialogues that would be usable for many different languages, as long as their data structures were extensions of VISUAL DISPATCHER's architecture. One main result of this effort are the dialogues for classes and methods, as they detect the possible attributes of an underlying language object and adapt themselves to those properties. Hence these dialogues are currently used for both Language Modes.
The last functionality that was missing from the JAVA Language Mode at that time was an engine that would do the simulation of the dispatching strategy. When beginning to implement such a component, an abstract class was used again to define all properties of such a "Language Handler". Using this approach it would be possible to make some assumptions on the structure of a concrete handler later on. But this was not at all the only problem to be solved. Since it makes no sense at all to dispatch on an incorrect class tree, because the outcome of such an event would not be defined by the language specification [2], there had to be a checking algorithm, which made sure everything was in accordance to the rules of JAVA. As this meant implementing a lot of rules stated in the JAVA Language Specification in a rather informal way, the lecture script [3] was of great help as it contained those rules in a much more formalized way. After having implemented those rules the actual dispatching algorithm wasn't that much work and as a side effect those on-line validity checks after each user interaction allow even deeper insight into the structure and mechanisms of JAVA.

1.4 Lootion Dispatching

When starting to work on the LOOTION dispatching engine, there was still little information available about the language's dispatching strategy. Nevertheless there were some facts that could be used to start implementing the required data structures. This was actually the first time concerns came up, if the framework could really support a completely different pro-

gramming language, because up to then everything had been quite focused on JAVA. But there were no troubles in succeeding and it was even possible to reuse most of the user interface classes, what can be considered quite a success.
Somewhere in the middle of the project the idea of using a LOOTION simulator, that was developed as another semester work, presented itself. As this would save a lot of work, after the experiences made with the JAVA counterpart of this class, it was decided to stick to this idea and wait for that dispatcher. Finally a presentation of both this and the other semester work was scheduled. During this presentation a lot of information about LOOTION became available and one could be sure, that the approach on LOOTION had been too naive. Without questions there were concepts that would not be possible to integrate without reworking a portion of the framework. As an example LOOTION uses concepts like “Rolesets” and “Conglomerate Classes” which really did not fit in any planned structure.
It was back to the drawing board. However, as far as data structures were concerned, the software was open enough to allow to be extended to support those new concepts. What was annoying were the requirements to the user interface. Integrating these concepts meant that each language must provide its own dialogues, menus and toolbars. To this end the structure of a “GUI Handler” was implemented. This class is used by the interface to integrate special properties of a given language and it has to provide some components discussed in a later chapter.
Having built this component and adapted all existing classes to the new requirements, it was no longer a problem to come up with highly customized menus and toolbars. And as an architectural feature there is nice separation between the handling of the data strucure (“Language Handler”) and the manipulation of the user interface (“GUI Handler”). At this point and up to now, one can be optimistic that there is no language that can’t be integrated into the framework of VISUAL DISPATCHER!
There is however one thing not to be particularly proud of. It was never possible to get a working LOOTION dispatcher and therefore it was never integrated into VISUAL DISPATCHER. The only thing documenting some LOOTION syntax was the specification of the input files of Thomas Amberg’s simulator. As a consequence an export module for this syntax was written, that one would be at least able to explore the world of LOOTION’s dispatching strategy in this less comfortable way. At least there is now a fool-proof and easy method to come up with such input-files and nevertheless VISUAL DISPATCHER does provide checking of all known language rules for LOOTION.

1.5 Special Features

This last section of the first chapter is dedicated to the features never planned or thought of in the first place. Surely one of these features are the on-line validity checks the JAVA dispatching engine performs. As an additional bonus the original compiler messages one would get if compiling such a project were taken to interact with the user.
The second additional functionality is the possibility to generate skeleton class files from a given design. Whereas in the case of LOOTION this is used to create a simulator input file, in the case of JAVA it can be used to export everything, compile it and test, if the virtual machine reaches the same result as VISUAL DISPATCHER. Or one could design his project using VISUAL DISPATCHER and the export it to implement the method bodies.

2 Using Visual Dispatcher

This chapter will give you a quick introduction in how to use VISUAL DISPATCHER. All features and functionality are explained using a step-by-step example of a typical session with VISUAL DISPATCHER.

2.1 Starting Visual Dispatcher

VISUAL DISPATCHER can be started as an applet or as an application. Depending on how the program is started several features will or won't be available due to applet restrictions. VISUAL DISPATCHER has been developed using version 1.2.2 of the JAVA Development Kit and has proven to run stable in this environment.

2.1.1 Starting Visual Dispatcher as Appliction

To start the application, a script called `startApp` (Figure 2.1) is provided in the main dircetory of the VISUAL DISPATCHER distribution. The script sets the classpath and launches the program.
To invoke the script change into the main directory and type `./startApp`. During the startup of VISUAL DISPATCHER the script will print out some information about the virtual machine and the class path. Make sure that the classpath is set in accordance to your system and that the JAVA virtual machine (`java`) is included in your path. If something fails while starting the application, this information should be used to determine the cause of error!

2.1.2 Starting Visual Dispatcher as Applet

If you would like to start Visual Dispatcher as an applet you can either use the program `appletviewer` which is part of the JAVA Development Kit (JDK) or you can install the distribution on your webserver. Note however that a client who wishes to use VISUAL DISPATCHER must have the JAVA SWING browser plugin installed in order for VISUAL DISPAT-

```
#!/bin/bash
APPHOME=‘pwd‘
JAVAVM=‘which java‘
echo
echo "Starting Visual Dispatcher 1.00"
echo "------------------------------"
echo
echo "User Command Line:         $0"
echo "Visual Dispatcher Classes: $APPHOME/classes"
echo "Java Virtual Machine:      $JAVAVM"
echo
$JAVAVM -version
echo
CLASSPATH="$CLASSPATH:$APPHOME/classes/"
$JAVAVM -classpath $CLASSPATH vdispatcher.VDApplet
```

Figure 2.1: startApp script

CHER to work.
The subdirectory `classes` contains a file `index.html` which shows how VISUAL DISPATCHER is to be integrated into a website. This file can also be used to start VISUAL DISPATCHER with the `appletviewer` application. To do so, change into the `classes` subdirectory and execute `appletviewer index.html`.

2.2 The Main Window

After starting VISUAL DISPATCHER the main window (Firgure 2.2) will be displayed on the screen. On the upper border of the window all functions of Visual Dispatcher can be accessed using either the menu bar or the toolbar below it. While the menubar contains all features, of the program the toolbar only provides quick access to the few often used ones. The center part of the window is occupied by the so-called “workpane”. When designing a class tree hierarchy this area will display all elements in an esay-to-understand graphical way. At the bottom of the window there is a status bar informing the user about the current state of the program.
There are two more features of the main window not shown in Figure 2.2. Those are the Console and the Viewers panel which can be displayed at the bottom of the window using the `View` menu from the menu bar.
The Console gives you a deeper insight in what the system is doing at any given time. If there is a problem with the class tree you designed error messages will be diplayed on the Console in real-time. When you finally start to dispatch methods on your hierarchy the result of the process will also be shown on the Console, together with some additional information documenting how the result was found.
When your design gets bigger the workpane will probable grow to small to show all classes at once. In order to help the user find specific elements of his class tree the Viewers pane was created. This list contains all object of the design on the workpane and by clicking on the

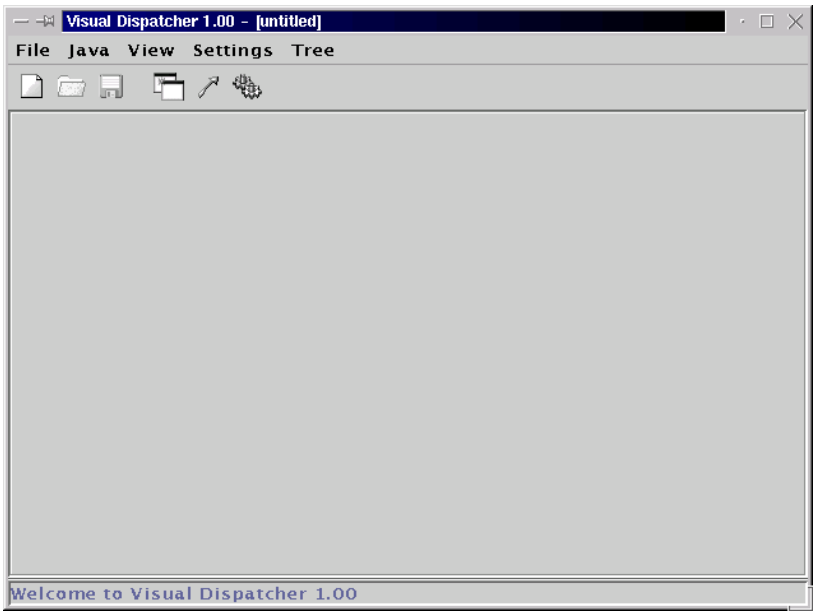

Figure 2.2: Visual Dispatcher Main Window

object's name it can be scrolled into focus!

2.3 Creating a Class Tree

To be able to study the different dispatching strategies used in object-oriented programming languages, one has to build an exemplary class tree first. This section will talk about the possibilities VISUAL DISPATCHER has to offer to achieve this task.

2.3.1 Creating a Class

Classes as a prototype description of objects can be found in every object-oriented programming language. As they are a central element of those languages, they are usually the first thing, one designs when creating a new class tree with VISUAL DISPATCHER.
To insert a new class into the current design one can choose the menu `Create Class` from the Language Menu. The name of the Language Menu varies according to the preset Language Mode. For the cause of simplicity we will conduct our example in the JAVA Language Mode, hence the name of the Language Menu is `Java` (Figure 2.2). As an alternative one could also create a new class by clicking on the corresponding button on the toolbar.
To create a class, a dialogue window (Figure 2.3) will appear on the screen. This window is partitioned into two regions. The upper region lets the user specify certain class properties as the name and modifiers. Note that the content of this region depends on the current Language Mode. Below this region there is another area which provides the possibility to state the superclasses of the new class. This list contains all classes in the current design. A class can be made a superclass of the new one by highlighting its name in the list. If multiple inheritance is supported by the Language Mode, one can also select multiple classes using the `CTRL` or `SHIFT` modifier keys when selecting the classes.

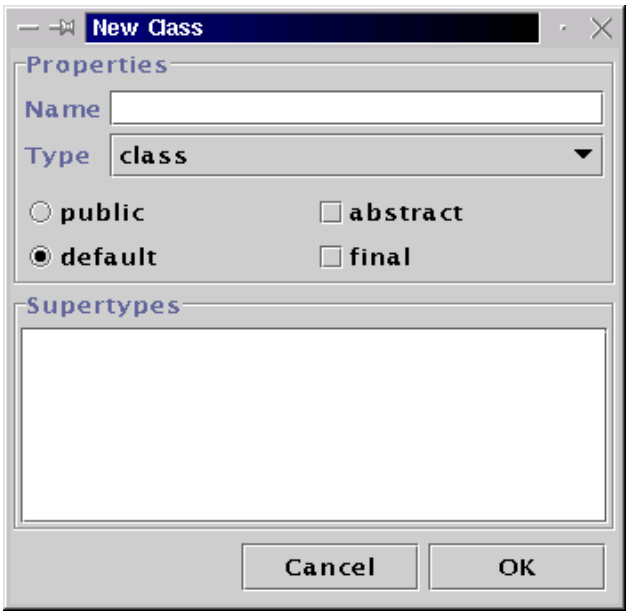

Figure 2.3: New Class

Figure 2.4: Class Viewer

After confirming the settings by clicking on the `OK` button, a little window (Figure 2.4) will appear on the workpane representing the newly created class. This window can be dragged to any position on the workpane. With a little luck it can also be resized by moving the mouse over the border of the window and dragging it to its new size.

2.3.2 Creating Subclasses

One very important concept of object-oriented programming languages is the notion of inheritance. An object can inherit certain properties and methods from another object. The classes serving as prototype for those objects are said to be in a subclass relationship.
VISUAL DISPATCHER provides two alternatives to establish such subclass relationships. One we have already seen: when creating a new class there is the possiblity of specifying the superclasses of the new class. Another way is the `Create Constraint` menu from the Language Menu. Again this function may also be accessed by invoking the corresponding button in the toolbar.
The dialogue shown in Figure 2.5 will then appear on the screen. Here the user can define one subclass relationship at the time by selecting the sub- and the superclass(es) to be associated together. If the Language Mode allows multiple inheritance more than one class can be selected on the right side by using the `CTRL` or `SHIFT` modifier keys.
Through clicking on the `OK` button the new constraint will be inserted into the class tree.

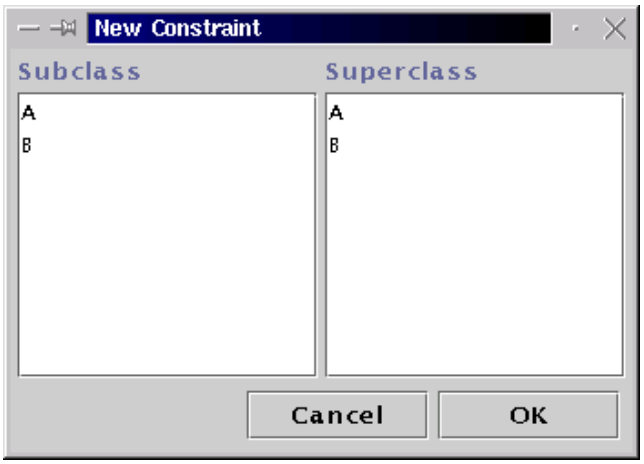

Figure 2.5: New Constraint

The subclass relationship is viewed by VISUAL DISPATCHER as an arrow pointing from the subclass to the superclass. Constraints cannot be resized or dragged around on the workpane. Their positon and layout is automatically determined based on the position and size of the sub- and superclass viewer.

2.3.3 Inserting Methods

Experimenting with dispatching strategies would not make very much sense without methods and method overloading. Respecting this fact VISUAL DISPATCHER lets the user define and insert methods into classes.
To do so, one selects the `Create Method` menu from the Language Menu or invokes the corresponding button on the toolbar. It is however important to select the class to which the method should be added first. If no class is selected there will be an error message!
A dialogue window (Figure 2.6) with all possible options and settings for methods will be displayed on the screen. As for classes the upper area of the window is used to set the name, return type and modifiers. The options displayed here are again dependent on the Language Mode.
Below there is an aera for adding parameters to the method. A parameter is a tuple consisting of a name and a type. The order of the parameters is in most languages an important property of the method's signature. Honouring this VISUAL DISPATCHER lets you order the parameters in an easy way, using the `Up` and `Down` buttons to move the selected parameter within the list. Parameters can be deleted using the `Remove` button.
Clicking on `OK` will insert the method into the previously selected class. The viewer of the class will then contain the head of the new method (Figure 2.7).

2.4 Dispatching Methods

After creating a design it is possible to dispatch methods on the class tree to evaluate, which methods will be called by the runtime environment of the object-oriented programming language.

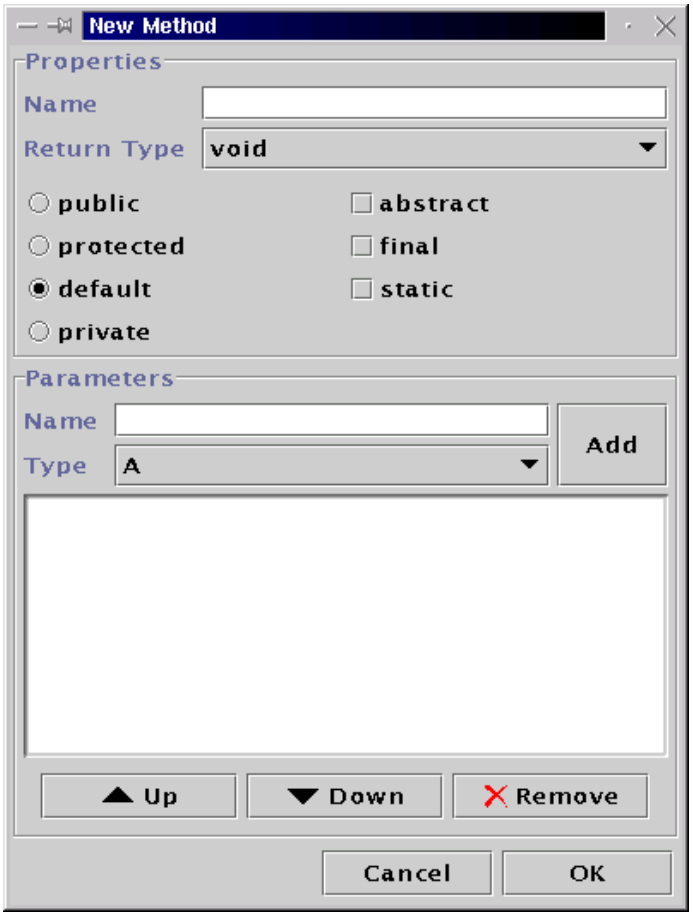

Figure 2.6: New Method

Suppose one created the class tree shown in Figure 2.8. What would happen if somewhere in `class B` there is a call `m(0)` to method `m`? VISUAL DISPATCHER can answer that question by simulating the behaviour of the chosen programming language. To start such a simulation select the `Dispatch Method` menu item from the Language Menu.
The appearing dialogue can be used to enter the question stated above. As it is not entirely intuitive, the main options are explained here.

Base Type This pulldown menu is used to specify the class from which the method call is made. One can think of the base type as the file in which the source code containing the call is stored. This information is only used to check the visibility of a given method in a given class.

Target Type This it the prototype class of the of the receiving object. For instance if the call is of the form `B.m(0)` then `B` would be the target type. However if the call is of the form `m(0)` then the target type is the same as the base type.

Visible Methods Based on the target type a set of visible methods is calculated on-line. Those methods need not be accessible from the base type! For instance if the class of the target type contains methods classified as `private` these methods will be included

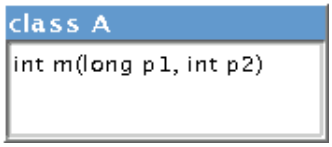

Figure 2.7: Class Viewer with Method

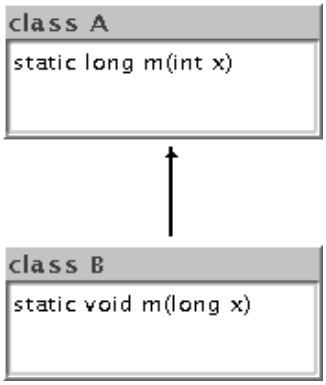

Figure 2.8: Example Design

in the list even if they cannot be called from a base type class other than the target type's one.

Note further that overidden and shadowed methods will be displayed once for every declaration along the inheritance chain. In the example, method `m` is declared in both `class A` and `class B`, hence it is included twice in the list. It is important to understand, that by selecting an item from the list, one rather selects a method signature than a concrete method.

Parameters Based on the selected method signature a list of parameters is generated. Here one can specify the statically declared type of the variables given as parameters to the method call. There is no on-line check if the specified type is at all valid compared to the method signature. This check however will be performed later by the dispatching engine.

After having specified all options in accordance to the call one wants to simulate, the dispatching progress can be invoked by clicking on the `Go!` button.
The system now evaluates the given situation and decides which method should be called. Depending on the selected Language Mode the procedure may be diffenrent. In our JAVA example, the system calculates the set of applicable methods first. From this set the most specific method is selected. If there is no most specific method, the method call is ambiguous and no method will be selected. In our example the following output can be observed on the programs Console.
Both methods are applicable since none is declared `private`. Since `m` is called with parameter `0` of type `int` the dispatcher will not be able to determine if it should call `m(long x)` or `m(int x)`. Since `B` is a subclass of `A` and `int` is a subset of `long` the two methods are not comparable and therefore there is no most specific method for `m(0)` (`Java` Language

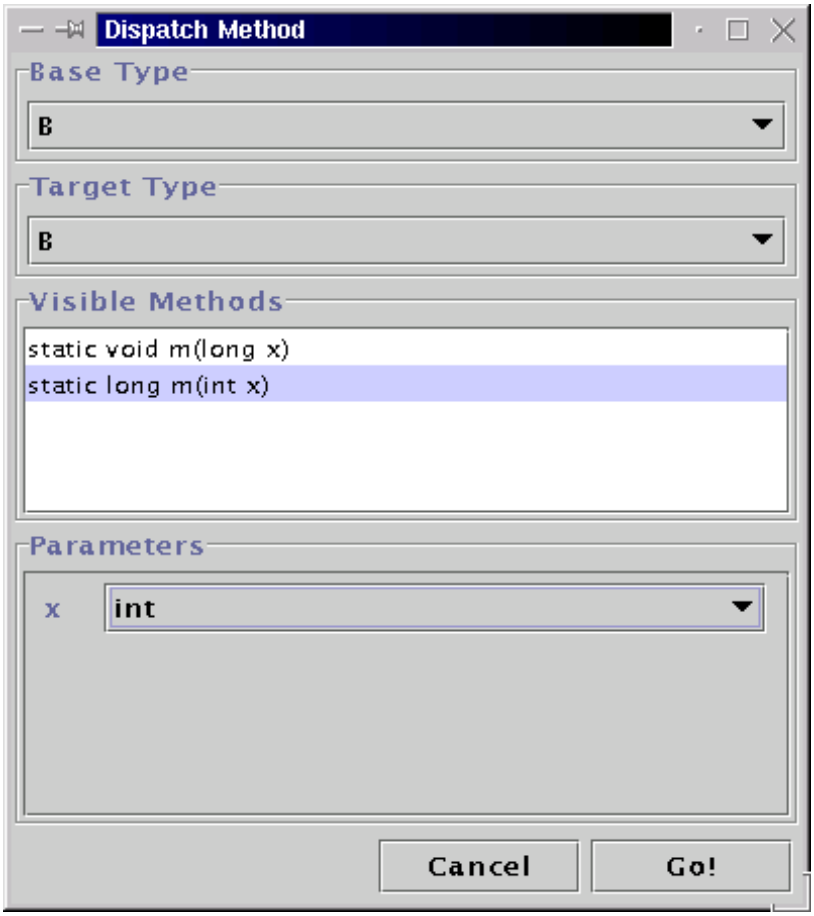

Figure 2.9: Dispatch Method

Specification [2], Chapter 15.11).

2.5 Further Functionality

2.5.1 Switching the Language Mode

As VISUAL DISPATCHER can be extended to support several programming languages, there is a menu item `Language` in the `Settings` menu. This menu item can be used to switch between all currently installed Language Modes. Keep in mind that switching the Language Mode will create a new design, since the various languages are not always entirely compatible with each other.

2.5.2 Saving and Loading Designs

Once a design is completed it can be saved using the `Save` menu item from the `File` Menu. VISUAL DISPATCHER saves the entire class tree, as well as the layout and the selected Language Mode. Note that existing files will be overwritten without prompting!
Later on saved designs can also be reloaded and changed. To do so choose the `Open` menu

```
Applicable Methods
    - static void m(long x)
    - static void m(int x)
Reference to m(int x) is ambiguous.
```

Figure 2.10: Example Output

item from the `File` menu. VISUAL DISPATCHER will then restore the class tree and its layout and adjust the Language Mode to the one stored in the file.

2.5.3 Exporting a Design

Quite an important feature is the possibility to export a class tree into skeleton files of the given programming language. This allows to design a whole application, check its design and then export it to files which are only missing the method implementations. To export a class tree choose the `Export` menu item from the `File` menu.

3
Software Documentation

This chapter is intended as a reference containing all material necessary to understand and extend the design of VISUAL DISPATCHER. In each section one particular data structure or concept will be explained. An overview over the complete architecture can be found at the end of this chapter in figure 3.3
Usually these sections begin by explaining why a certain design was used. Then there is the documentation of the classes used, together with their respective interfaces, followed by material showing, how to use and extend these concepts to integrate further dispatching strategies.

3.1 The Class Tree Data Structure

A central element of Visual Dispatcher is the data structure called class tree shown in Figure 3.1.

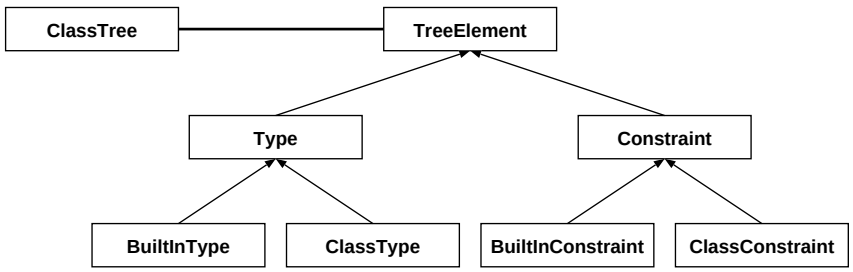

Figure 3.1: Class Tree Data Structure

This data structure is used to store information about the classes and subclass constraints, the user is entering via the graphical user interface. Later on this information is used to perform validity checks and to finally simulate the dispatching of methods. It is therefore important to store this data in a smart and easy-to-access way. As a consequence the class tree data

structure consists of two major components. On of these components is class `ClassTree`, the other is the class hierarchy with root `TreeElement`.

3.1.1 Class `ClassTree`

Class `ClassTree` implements an Abstract Data Type (ADT) for the class tree by means of a heterogenous double-linked list. Its interface operates on elements of type `TreeElement` and can therefore accept and store all objects inheriting from class `TreeElement`. The following is a description of the most commonly used methods of class `ClassTree`.

`public ClassTree()` A standard constructor that will generate and return a new and empty class tree. No parameters need or can be specified.

`public void insert(TreeElement e)` Once the class tree is instantiated elements of type `TreeElement` can be inserted. Note that this can be any object inheriting from class `TreeElement`. There is no check performed if the element in question is already contained in the class tree data structure. If this is the case it will be inserted again!

`public void remove(TreeElement e)` Removes the given element. Again no checking, if it is at all contained in the class tree, is performed.

`public void reset()` The data structure implements an internal pointer to a so-called `current` element. This is used to iterate through the list. By calling `reset()` the pointer is set to the `first` element contained in the list.

`public boolean hasMore()` Returns true if there are more elements contained in the data structure following the `current` element. This can be used to implement loops iterating on the list.

`public boolean hasLess()` Analogous to the previous method but this one returns true if there are more elements contained in the list preceeding the `current` one.

`public TreeElement getNext()` Returns the next element in the class tree after the `current` one. As a side-effect it increments the `current` pointer.

`public TreeElement getPrev()` Returns the previous element with respect to the `current` one. As a side-effect it decrements the `current` pointer.

`public Vector getAll()` Returns a vector containing all elements contained in the class tree. This method can be used as a reference on how to implement an iterator over the class tree data structure.

As for now, I would guess that this class is more or less complete and will not need to be modified or extended to support new dispatching paradigms or new object-oriented programming languages. The only thing that could be changed is the internal storage of the class tree while keeping the current interface intact. I have chosen a linked list in favour of a more complex data structure, because I felt that for the given problem sizes there will be no great performance loss.

3.1.2 `TreeElement` Class Hierarchy

At the moment the `TreeElement` class hierarchy consits of seven different objects. Basically there are but two really different objects, that can be stored in a class tree, objects of type `Type` and `Constraint`. Both objects are implemented as abstract classes and have no unexpected or particularly interesting properties.
On the next level of inheritance (Figure 3.1) these two classes are further specialized into two separate classes each. For both `Type` and `Constraint` there exists a built-in and a class version of this object. These classes had to be introduced to support types and constraints that are not integrated into the object-oriented model of a given programming language.

Class `BuiltInType`

This is a very simple concrete class. No properies of the ancestor `Type` are extended or overidden. Class `BuiltInType` serves merly as a way of distinguishing class types from built-in types. A very good example of a possible use for this class are for instance types like `int` or `boolean` in JAVA.

Class `ClassType`

In contrary to the class discussed above, `ClassType` is indeed extending its ancestor. This class adds fields to store modifiers, class variables (unused) and most important methods. It implements "setters" and "getters" for all of these fields. As the classes of a given programming language may contain further properties and different modifiers this class is declared abstract and has to be specialized into a concrete class. When doing so, three methods have to be implemented to manage the different kinds of modifiers.

`protected void initPropertyModifier()` Property modifiers usually specify if a class is abstract or not.

`protected void initAccessModifier()` Access modifiers control access to a given class. In JAVA for instance four different level of access control can be set.

`protected void initTypeModifier()` Type modifiers are a bit an exotic construct. I had to use them, because in JAVA a class type can also be an interface, a kind of restricted class to support limited multiple inheritance. Pure object-oriented languages won't have to use this feature.

These three methods are then called in the constructor to initialize the modifiers. A possible implementation of these methods can be seen for instance in class `JavaClass`. How modifiers are modelled as objects in VISUAL DISPATCHER and how they are used is the subject of a later section.

Class `BuiltInConstraint`

As the class for built-in types, this class has nothing really interesting to offer. Again it is rather used for classification than for specialization. It does however restrict contraints to pairs of type `BuiltInType`, as it would be useless to associate other types with built-in types. Further it is important to understand, that built-in constraints do not model inheritance

relationships between the types they connect. Rather do they express the fact, that the domain of one type is a subset of the domain of another type. An example for this would be the types `int` and `long` in JAVA.

Class `ClassConstraint`

Objects of type `ClassConstraint` are used to model inheritance in the class tree. One object represents one sub- and superclass tuple. Even if the language supports multiple inheritance there is always one constraint for each ancestor. This class too is a very simple class not really extending class `Constraint` but serving as a mean of classification and restricting constraints to pairs of type `ClassType`.

3.2 The Class Tree Viewer

As the class tree data structure discussed in the previous section is the internal representation of the class tree, the class tree viewer is used to display this structure in the graphical user interface.
First of all it is noteworthy, that the viewer component is much simpler in its design than the underlying data structure. The reason herefore is that built-in types and constraints are not viewed in the program and serve for algorithmic purposes only.

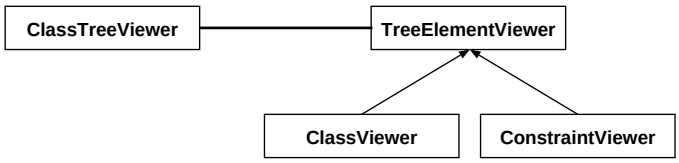

Figure 3.2: Class Tree Viewer

When looking at the design shown in Figure 3.2 we note the partition of the viewer into two different structures.
On the one side there is class `ClassTreeViewer` a GUI component which can be embedded into SWING-based applications, on the other hand there is the class hierarchy with root `TreeElementViewer` providing viewer components for the two basic concepts the program has to be able to display.

3.2.1 Class `ClassTreeViewer`

Class `ClassTreeViewer` is a full-featured SWING component. It is a extension of `javax.swing.JLayeredPane` and provides some additional functionality to manage objects of type `TreeElementViewer`. `JLayeredPane` was chosen as an ancestor, because it implements a layer concept which helps to implement dragging easily. Also, it is important to have a possibility to position constraint viewers behind class viewers in order to get a nice layout on the screen. In the following paragraphs some of the additional features of this component are documented.

`public ClassTreeViewer()` Generates a new instance of a class tree viewer. The component manages a pointer `activeViewer` to the currently selected viewer which

is initialized to `null` in the constructor.

`public void insert(TreeElement e)` It has to be possible to insert new objects into this container. Although this could be done using the `add` method inherited from `JLayeredPane`, there is this special method which takes a object of type `TreeElement` as parameter. Based on the type of the parameter the appropriate viewer is generated automatically and inserted into the container at a default position.

`public void remove(TreeElement e)` This method removes a viewer from the container based on the object given as parameter. If there is no viewer matching the given object nothing will be done.

`public void setActive(TreeElement e)` Marks the viewer for the given object as selected. The previously active viewer will be deactivated.

`public TreeElement getActive()` Returns the object of type `TreeElement` which is associated with the active viewer. If no viewer is active this method returns `null`.

To be able to synchronize this component with the surrounding user interface, several property change messages are fired during operation.

viewerInserted Fired by `insert` whenever a new viewer is inserted. As example this event is used by the user interface to update the list of all displayed viewers.

viewerRemoved Analogous to the event viewerInserted this one is fired by the methods `remove` and `removeAll` whenever a viewer is removed.

viewerActivated This event is fired when a viewer gets the focus because the user clicked on it. Together with the event, the newly selected viewer is broadcasted.

To interact with events comming from the viewers contained in this component, there is an internal class implementing a property change listener. This listener is added to every viewer on creation. It reacts in the following manner to three events.

viewerStateChange This event is received when the state of a viewer changes. This may be caused by resizing or moving a viewer and causes the container to repaint itself.

focusStateChange A viewer inside the container will fire this event if the user clicks on it. This triggers the component to change the selected viewer and fire the viewerActivated event.

viewerDragged Whenever a viewer is being dragged around it will fire this event. The container will then lift that viewer to the top layer of the layered pane. This allows the user to drag viewers over other components in the container.

3.2.2 `TreeElementViewer` Class Hierarchy

The `TreeElementViewer` class hierarchy consits of three classes. The most important is without doubt the abstract class `TreeElementViewer` itself. This class implements some very basic concepts, that can be used by the inheriting classes. For instance this class implements the complete mouse handling including dragging and resizing. To be able to do so, this class has to know a lot about its concrete instances. Therefore a set of abstract methods have to be implemented by inheriting classes.

`public boolean isDraggable()` True if the component should be draggable for the user. This is for instance to distinguish between constraint and class viewers.

`public boolean inDragArea(Point loc)` True if the mouse position `loc` is in an area on the viewer where it should be possible to drag the viewer.

`public boolean isResizeable()` Analogous to `isDraggable` this method returns true if the viewer in question is resizeable.

`public boolean inResizeArea(Point loc)` This is the same as `inDragArea` only this method returns true, if the mouse is at a position where it should be possible to resize the viewer. Typically this would be the borders of a component.

`public int getResizeDirection()` During the resizing process this method returns the direction in which the viewer is being resized.

`public Cursor getCursor(Point loc)` This method returns a mouse cursor to a given mouse position. This is merly a question of software usability as the changement in cursor indicates what possibilities a user has a any given position.

`public void initComponent()` This method is used by the `ClassTreeViewer` class to perform an initial setup of this viewer after it has been created. It therefore should setup the size and the components contained in the viewer.

`public void update()` Indicates to the viewer that the element it is viewing has changed. The method has to update the displayed viewer to the actual state of the element.

A good example of how to implement these methods, can be taken from one of the classes `ClassViewer` or `ConstraintViewer`. Nevertheless I feel, that it probably won't be necessary to implement additional viewer types.

3.3 Methods and Signatures

When planing on simulating the behaviour of a specific dispatching strategy, there has to be a way to store methods for each object representing a class. But this is usually not enough. Programming languages such as JAVA use a concept called signatures to compare one method with another. VISUAL DISPATCHER provides two abstract classes (`Method` and `Signature`) to model these facts.

3.3.1 Class `Method`

Class `Method` is used to model methods of an object-orieneted programming language. Methods have a certain set of properties such as a name, a return type, a collection of parameters and possible modifiers. For all these concepts, this class provides adequate storing facilities. Two less obvious fields are the one where the declaring class is stored and the one which stores a string representing the void return type.
During the simulation of dispatching mechanisms it is often important to know which methods are possible candidates. To evaluate this set one has to know in which class a method was declared hence the field `declaringClass`. The other less intuitive field `voidReturn` was introduced to be able to specifiy any possible keyword to symbolize that the method has no return value. In JAVA and C++ this keyword is `void`, but in other future languages it could be something completely different. As it is neither a built-in nor a class type, I've decided to model it as a special property of class `Method`.
Class `Method` is an abstract class and to use it, it has to be extended. There are several methods that need to be implemented by an extension of this class. As with class `ClassType` there are methods to initialize some properties in a customized way. Then there are two more methods dealing with the issue of signatures.

`protected void initPropertyModifier()` Sets the property modifiers in accordance to the language specification of this method object. A example for a property modifier in JAVA would be the keyword `final`.

`protected void initAccessModifier()` This method deals with the access modifiers. If you need an example, think of things like `public` in JAVA.

`protected void initVoidReturn()` Initializes the string that should be displayed, when a method has no return type.

`public Signature getSignature()` Computes and returns the signature of this method. This can be different from one object-oriented language to another. The same functionality has to be implemented using a set of parameter instead of the internal fields of the class.

As a reference on how to extend class `Method` the reader can have a look at class `JavaMethod`, which implements an object representing a method within the JAVA Language Mode of VISUAL DISPATCHER.

3.3.2 Class `Signature`

As mentioned before a virtual machine does ususally not work with all properties a method has to offer. Commonly a signature of the method is used to compare one method to another. If you are new to this concept, think of the signature as a subset of all the properties a method object has. In JAVA the signature of a method includes the name of the method, the order, number and types of the parameters. It does however not include things like the return type or the names of the parameters.
As there can be made absolutely no assumptions on what is contained in a signature, class `Signature` does not have any fields. The only thing it requests from an inheriting class is the implementation of one abstract method.

`public boolean compare(Signature s)` Returns true if the given signature `s` matches the current one.

Again reference material can be found in the form of a concrete implementation of such a signature object in class `JavaSignature`.

3.4 Modifiers and Parameters

In the last sections two concepts were talked about, without really ever explaining what they are and more important, how they are being represented in VISUAL DISPATCHER. Those concepts are modifiers and parameters and there are two classes which model them as JAVA objects. For the lack of better names these classes are simply called `Modifier` and `Parameter`. As they probably won't have to be extended there is just a quick description on how to use them.

3.4.1 Class `Modifier`

Modifiers are used in object-oriented programming languages to state global properties of a class, a method or a field. In JAVA for instance access modifiers such as `private`, `protected` and `public` can be put in front of a method declaration to limit access to this method. From this short description one quickly understands that there have to be at least two properties in a class representing a modifier: the name of the modifier and a state indicating whether it is set or not.
Matters become just a little more complicated as it is pefectly legal to state no modifiers at all. In the case of access modifiers in JAVA, this represents default access to an object. To display this "default" option in the graphical user interface the name of the modifier will have to be set to `default`. But now a problem arises when one tries to display the method in the syntax of the chosen programming language, because `default` is usually no keyword of the language. To solve this problem, there is a third field `printName` that allows the user to set a string that should be used whenever the modifier is being displayed in the syntax of the programming language.
The interface of class `Modifier` has nothing really interesting to offer. As one can expect, it mainly consits of "setters" and "getters" for the three discussed fields and it overrides `toString()` to behave as described above.

3.4.2 Class `Parameter`

Parameters are tuples of a name and a type. Knowing this the implementation of class `Parameter` will not astonish anyone. The main purpose of this class is to augment the readability of the code.

3.5 LanguageHandler and GUIHandler

As of now it should have become clear, how VISUAL DISPATCHER stores and manages the data it uses to simulate the dispatching of methods. What has not been talked about however,

is how the different properties of various object-oriented programming languages are intergrated into the software. When thinking about this problem one sees, that the challenge is really twofold. On the one hand there are dispatching algorithms that differ from one language to another. On the other hand the graphical user interface has to be flexible enough to allow integration of custom menus and toolbars.
Honouring these aspects, I have divided these responsibilities into two separate classes. The class `LanguageHandler` is concerned with all that regards the language itself. Integrating different functionality on the other side is the task of class `GUIHandler`. Both classes are abstract classes and define a set of abstract methods that have to be implemented by an inheriting class.

3.5.1 Class `LanguageHandler`

As mentioned before, the task of class `LanguageHandler` is to manage all aspects of the represented programming language. This is needed in order to be able to access this functionality from the user interface in a transparent manner. To do so, there are three thematic groups of methods. Whenever possible, there is a default implementation. If no such implementation could be found or would make sense, the methods are kept abstract and will have to be implemented by any concrete class modelling a specific language handler.

Object Creation

The methods in this group deal with the creation of language specific objects. These methods are called by the graphical user interface whenever the user requests an new instance of a given object.

- `public ClassType createClass()`
- `public ClassType createClass(String name)`
- `public Constraint createConstraint(ClassType s, ClassType t)`
- `public Method createMethod(Type declaringClass)`
- `public Method createMethod(String name, Type declaringClass)`

Object Checking

There should be an easy way to check if the design the user is fabricating is at all valid. The methods in this group provide just this functionality. Using them the user interface can check the users actions at any given granularity.

- `public boolean checkClass(ClassType classObj)`
- `public boolean checkMethod(Method methodObj)`
- `public boolean checkTree()`

- `public boolean checkCycle(ClassType base)`
- `public boolean checkName(String str)`
- `public boolean checkClassName(String s, ClassType c)`
- `public boolean isAlpha(char c)`
- `public boolean isAlphaNum(char c)`

Language Features

Usually during dispatching, some specific sets of methods will be computed in a way specific to the programming language modelled. These sets are the visible methods, the applicable methods and the most specific method. To compute those sets other functions are usually employed to determine if one function is overriding another or if a method is at all accessible. The methods in this group will allow you to do just that.

- `public boolean overrides(Method meth1, Method meth2)`
- `public boolean isAccessible(ClassType classObj, Method meth)`
- `public Vector getVisibleMethods(ClassType classObj)`
- `public Vector getApplicableMethods(ClassType c, Signature s)`
- `public Method getMostSpecificMethod(Vector applicables)`
- `public boolean subtypes(Type subType, Type superType)`

Class `JavaHandler`, which implements these methods in accordance to the JAVA Language Specification [2, 3], provides a very good example of how to build such a “Language Handler”.

3.5.2 Class `GUIHandler`

From its structure the `GUIHandler` class is much simpler than the “Language Handler”. Basically there are just three abstract methods that need to be implemented by a concrete handler. These three method aim at integrating the features needed by different programming languages into the graphical user interface.

- `public JMenu getMenu()`
- `public JToolBar getToolBar()`
- `public JPopupMenu getPopupMenu(TreeElement e)`

The first of these functions should return a menu with the name of the programming language and all operations that can be performed on the class tree. This menu will be integrated in the menu bar of the main window in between `File` and `View`. Method `getToolBar` returns a toolbar with the most commonly used functions of this Language Mode. The toolbar is then shown at the top of VISUAL DISPATCHER's main window. The third method finally is used to attach a customized popup menu to every object in the class tree on the workpane.
Altough this structure is fairly simple, one must not forget that it is quite time consuming to implement a fully featured handler. For every item in a menu or on a toolbar there has to be an action listener or an abstract action. Often further dialogues and windows will have to be implemented to request user input. I tried to build most dialogs in an easy to reuse manner and many of them can adapt to the Language Mode automatically. Nevertheless the package for JAVA consists of six additional classes to the class `JavaGUIHandler`!

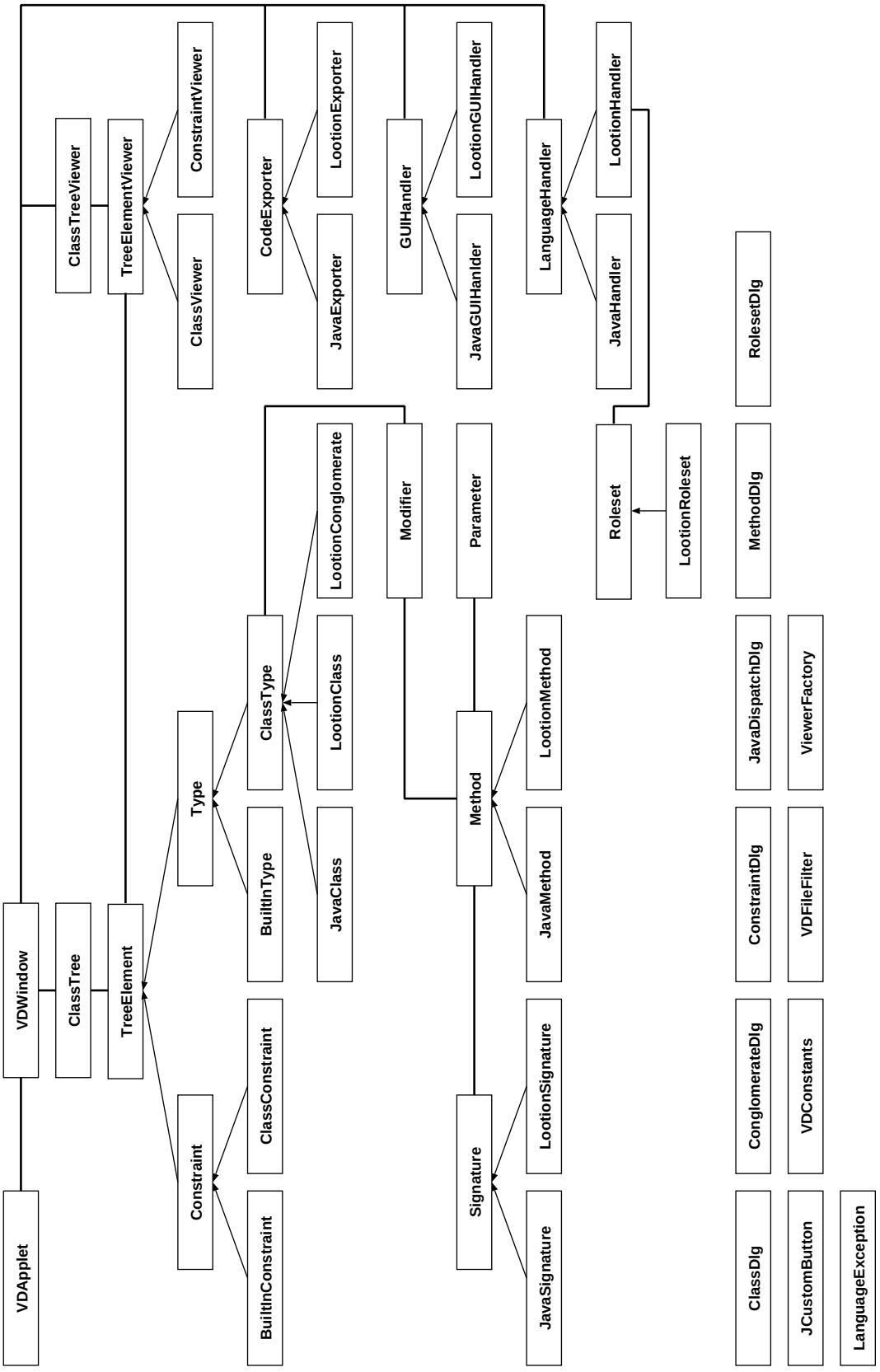

Figure 3.3: Complete Structure of Visual Dispatcher

4 Future Work

While working on VISUAL DISPATCHER there were some ideas, that couldn't be integrated into the program for the lack of time. Furthermore there are aspects of Lootion not known at this time. The LOOTION dispatching engine is therefore far from complete. This chapter will talk about three main issues that could be worked on in the future. All these topics are very interesting and implementing them would help to make VISUAL DISPATCHER an even better and more useful software.

4.1 Lootion Dispatching

As mentioned in this chapter's introduction the LOOTION programming language is currently under development. When working on VISUAL DISPATCHER, there were but a few bits of information about the syntax and features of LOOTION. As a consequence more focus was given to the JAVA part of VISUAL DISPATCHER.
At the moment there is no intergrated dispatching engine for LOOTION. It is however possible to export a complete LOOTION design as a file understood by Thomas Amberg's simulator. To really experiment with LOOTION and to evaluate its dispatching behaviour this solution is far from being satisfying. One major task is therefore to integrate LOOTION's dispatching strategies as soon as the dispatcher is completed.

4.2 Importing Existing Classes

When experimenting with the JAVA dispatching engine, it is not possible to use classes that are not defined in the current design. One quickly realizes that this situation is very far from reality, since it renders the use of common classes such as `String` or `Vector` as parameter or return types impossible.
One solution that comes to mind is to use JAVA's Reflection API to import such classes and display them inside the user's design. Of course this would mean, that a "placing algorithm"

would have to be built in, to display the imported classes neatly. One must also decide on how far to go with matters concerning inheritance. If some naive user is about to import a SWING component, is it really desirable to import all ancestors and interfaces? As this is serious work, it was decided not to implement it but rather leave as an open task. Nevertheless it would be an important and useful feature.

4.3 Integrated Development Environment (IDE)

The third and last idea on how to extend or improve VISUAL DISPATCHER is a little bit more ambitious. When working on a project one learns to appreciate the comfort, an Integrated Development Environment (IDE) has to offer! Although there is much debate on how to design object-oriented applications in terms of structure and when to use which design pattern [7], none of the IDEs offers the possibility to design a project in a graphical way. Sure one can always use tools like RATIONAL ROSE, but there is no software combining application design with application development.
VISUAL DISPATCHER at it current state, provides a powerful framework for designing and testing the structure of an application. Therefore it would make sense, to extend the software to be able to manage code, compile, run and debug projects. On the one side this could be a very interesting project in terms of application programming and data structures, but it would also offer the possibility to provide an easy way to program in new languages such as LOOTION, thereby promoting it amongst users!

Acknowledgements

Last but not least I would like to mention some people, who helped me work on this project in some way or another. Best thanks to all of them!

Christian Bach Being the creator of LOOTION he gave me a wonderful and interesting opportunity to work on something completely new and unknown to me. I profoundly hope my work is in any way supportive to his.

Moira Norrie Supporting me in whatever I want to do, she gave me the possibility to do just the semester work I whished for.

My Family Listening to me whenever I talk to them about my concerns and providing an optimal working environment at home, my family gives me great strength and power.

Alex de Spindler Alex has been a very good friend and my chats with him helped me to think about other things than programming.

Pedro Gonnet Sometimes it helps to talk to somebody about what one is doing at the time. Pedro was always there to listen or discuss and provided helpful feedback.

Noémie Lerch Writing too is a very lonesome task. I therefore greatly enjoyed every possible opportunity to switch off the computer. Such an opportunity was daily lunch with Noémie.

Bibliography

[1] C. Chambers. Object-Oriented Multi-Methods in Cecil. Department of Computer Science and Engineering, FR-35, University of Washington, Seattle, WA 98195.

[2] J. Gosling, B. Joy, and G. Steele. The JAVA(tm) Language Specification. Addison-Weseley, Reading, Massachusetts, 1996.

[3] R. Stärk. Abstract State Machines for JAVA Lecture Script for Theoretical Computer Science (37-402), ETH Zurich, 1999.

[4] D. Flanagan. JAVA in a Nutshell. O'Reilly and Associates, 1996.

[5] D. M. Geary. Graphic JAVA - Mastering the JFC, Volume II: SWING. Sun Microsystems Press, Palo Alto, California, 1999.

[6] C. Bach. LOOTION (Language Promoting Object-Oriented Thinking and Software Construction) - Concepts. Diploma Thesis, ETH Zurich, 1999.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns - Elements of Reusable Object-Oriented Software. Addison-Weseley, Reading, Massachusetts, 1995.